MASTER EN SCIENCES INFORMATIQUES

Détection de codes sources fonctionnellement similaires

Munsi, Masia

Award date:
2019

Awarding institution:
Universite de Namur

UNIVERSITÉ DE NAMUR

FACULTÉ D'INFORMATIQUE

Détection de codes sources fonctionnellement similaires

MUNSI Masia Patrick

RUE GRANDGAGNAGE,21 ▮ B-5000 NAMUR(BELGIUM)

Résumé

Dans le domaine du génie logiciel, la recherche de codes similaires est une tache incontournable [33]. Elle s'effectue entre autres lors d'opérations de factorisation de codes, lors d'analyse d'impact, lors de l'identification de portions de codes qui ont été plagiés [5]. Dans des cas d'écolage, on y a recours pour la compréhension ou pour le développement de fonctionnalités. Les approches traditionnelles pour la détection de clones logiciels permettent d'identifier des clones de type textuel, syntaxique, structurel et métrologique, mais elles sont peu adaptées pour la détection des clones sémantiques [5]. Récemment, de nouvelles approches basées sur l'apprentissage automatique ont été proposées dans la littérature [17] [33] [22] [3].

Ce mémoire commence par exposer les approches classiques de recherche de clones logiciels. Puis propose une implémentation basée sur le plongement lexical. Cette implémentation est réalisée à l'aide d'un réseau de neurones et s'appuie sur un vaste jeu de données. Son objectif est de pouvoir détecter des codes sources Python sémantiquement similaires.

Mots-clés: Clone logiciel, Similarité sémantique, Plongement lexical, Apprentissage automatique, Apprentissage non supervisé, Réseau de neurones

Abstract

In software engineering, looking for matching codes can be crucial in several situations [33], such as software clones factorization, impact analysis and to highlight cases involving chunks of code that have been subject to plagiarism [5]. In educational situations, it is common to look for an equivalent functional portions of code for understanding and comprehension purposes. Several approaches have been developed in order to find textual, lexical, syntactic, structural and metrological clones, but these approaches are poorly adapted to finding semantic clones [5]. Lately, new approaches based on machine learning have been proposed in the literature [17] [33] [22] [3].

In this Master Thesis, I will start by a review of the traditional methods for finding functional similarities across different source codes. Then, an implementation based on word embedding will be developed. This implementation is achieved by using a neural network and relies on a large dataset. Its objective is to detect Python codes that are semantically similar.

Keywords: Software clone, Semantic similarity, Word embedding, Machine learning, Unsupervised learning, Neural network

Remerciements

Je tiens à remercier Messieurs Benoît Frénay et Benoît Vanderose, Professeurs à l'Université de Namur, de m'avoir guidé lors de l'élaboration de ce mémoire.

Je tiens également à remercier toutes les personnes ayant vécu, ces dernières années, ces choix à mes côtés. À mes garçons dont l'âge est inférieur à cette période, à ma compagne qui fut présente pour pallier à mes absences. Je remercie également Françoise et mes parents pour leur indéfectible soutien.

[french]babel

Contents

[british]babel

1 Introduction

Que ce soit lors de cycles éducatifs, dans des projets personnels ou dans la vie professionnelle, nous interagissons constamment avec des codes sources ayant été développés par d'autres personnes. Quand on désire mettre en place une nouvelle fonctionnalité, il est commun de s'inspirer et de se renseigner au préalable sur l'existence d'une fonctionnalité ou d'une librairie l'implémentant.

Dans le monde professionnel, par exemple, il est commun de reprendre des portions de code dont l'efficacité a déjà été éprouvée, menant parfois à des situations de *Code smells* tels que la multiplication de clones logiciels [5].

De nombreux outils pour la recherche de codes similaires ont étés proposés dans la littérature[31]. Récemment, de nouvelles approches basées sur intelligence artificielle (Machine learning) ont été proposées[17] [33] [22] [3]. Les techniques de *Machine learning* sont actuellement appliquées dans un large spectre de domaines tels que la finance, l'actuariat, l'oncologie, etc. Dans ce mémoire, nous envisagerons une approche de détection de codes similaires basée sur les réseaux de neurones.

Ce mémoire commence par un état de l'art présentant les différentes approches statiques pour la détection de similarités au sein de codes sources. Le chapitre suivant est une introduction aux réseaux de neurones implémentés via Doc2Vec. Le dernier chapitre est une application pratique permettant la détection de scripts Python fonctionnellement équivalents.

1.1 Les questions de recherche

Sur base de quels critères peut-on évaluer la similitude fonctionnelle de blocs de codes?

Dans quelle mesure les algorithmes de recherche de similitudes sont-ils applicables dans le cadre de blocs de codes implémentés dans un langage de programmation donné?

Quelle est la complexité de mise en place de ces algorithmes et quels sont les coûts en termes de ressources et de temps d'exécution résultant de l'utilisation de ces algorithmes?

Peut-on construire un outil utilisant ces algorithmes afin de détecter des similitudes au sein de projets implémentés dans un langage de programmation donné?

2 Les Clones logiciels

2.1 Définitions

Le terme "clone" peut être utilisé lorsque des éléments de similitude peuvent être trouvés entre des fragments de codes distincts. Actuellement, il n'y a pas encore de consensus et de définition générique ni sur le terme de "clone logiciel" ni sur le terme de "similitude". Des variantes de définitions sont employées dans la littérature et la définition suivante est celle sur laquelle on s'appuie communément.

"Clones are segments of code that are similar according to some definition of similarity [4]"

De cette définition, il en ressort tout de même quelques imprécisions:

1. Dans cette définition, on ne parle que de codes sources. Il serait intéressant de pouvoir tenir compte d'autres types d'artefacts tels que des modèles, des spécifications fonctionnelles [18].

2. La définition ne précise ni la taille des segments de code, ni la façon de procéder pour la définir, sachant que la taille des segments de code peut être définie statiquement ou dynamiquement (nombre de lignes, portée du code).

3. Par le morceau de phrase "Some definition of similarity", aucune précision n'est fournie sur les formes de ressemblances qu'il faudra détecter pour mettre en évidence les similarités recherchées.

De ce fait, bien que cette définition soit souvent présente dans la littérature, elle ne fait que préfacer plusieurs autres définitions et taxonomies.

Sur base des définitions existantes, on peut dégager des points communs. Les taxonomies existantes permettent de classer les clones en trois groupes. Le premier groupe est axé sur le type de similarité qu'il peut exister entre les clones. Le second est axé sur la localisation des clones. Le dernier groupe est axé sur la phénoménologie des clones et ce, dans le but d'effectuer du refactoring sur les codes clonés.

Le graphe (figure 1) permet d'avoir une vue générale de la classification des clones. De ce graphe, on peut en ressortir les clones des types 1 à 4.

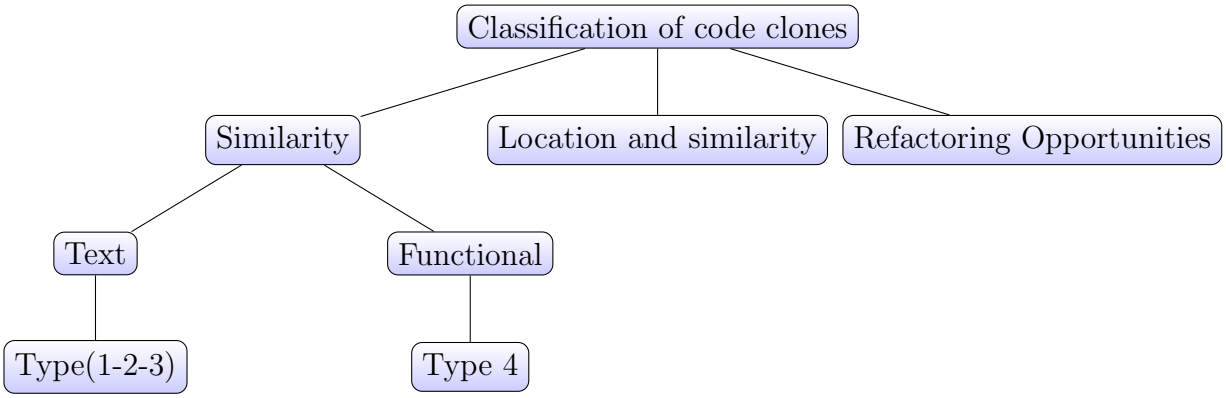

Figure 1: Classification des types de clone

Deux groupes peuvent être considérés. Le premier groupe comprend des clones dont les ressemblances sont propres au contenu lexical du code source. Le second regroupe comprend des clones dont les ressemblances sont liées aux fonctionnalités et aux comportements du code source. [11]

- **Type-1 (Clones exacts)** Au sein des clones de type-1, aussi nommés clones exacts, les différences entre le code originel et la copie sont au niveau de la mise en forme et de l'agencement du code. Les différences sont des changements mineurs de type esthétique tels que des ajouts d'espaces, de tabulations, des modifications de lignes de commentaire. On parle de clones exacts car les morceaux de code sont des copies à l'identique. Néanmoins, les méthodes de détection travaillant ligne par ligne ne détectent pas toujours ces simples modifications [16].

- **Type-2 (Renamed clones)** Dans les clones de type-2, il existe des différences entre les identificateurs. Les variables, les constantes, les classes et les fonctions peuvent être renommées. Les clones de type-2 sont structurellement et syntaxiquement semblables. C'est-à-dire que, pour deux fragments de codes de type-2, des substitutions d'identificateurs permettent de retrouver des clones de type-1 [18]. D'où l'appellation "Renamed-clones" [35].

- **Type-3** Dans ce troisième cas, il est toujours possible de trouver des similarités de type lexical. L'entièreté des déclarations et des instructions peuvent être modifiées. Ces modifications peuvent être des ajouts, des suppressions à plusieurs endroits du code source.

- **Type-4 (Clones sémantiques)** Pour les clones de type-4, le comportement fonctionnel des fragments de codes est observé. On s'intéresse aux similarités de type sémantique. Des fragments de codes effectuant les mêmes opérations (des préconditions et des postconditions identiques), mais ayant des implémentations différentes, peuvent être considérés comme clones.

Parmi les termes faisant référence à ce type de clone, on peut également citer les Clones locaux, Clones creux, Clones structurels, Clones fonctionnels.

2.2 Phase de détection de similarités

L'apparition de fragments de codes similaires est fréquente, surtout pour les vieilles applications en production de grande taille. Il est commun que des développeurs empruntent des morceaux de code provenant d'autres emplacements afin d'implémenter les nouvelles fonctionnalités. La détection de clones est pertinente dans la mesure où ce schéma de fonctionnement peut engendrer de la complexité structurelle impactant la compréhension, la maintenance et les performances du code [9].

Il n'est pas toujours possible de détecter les morceaux de code qui ont fait l'objet de duplications. Une comparaison doit être réalisée entre tous les fragments de codes, aussi bien du côté du code originel que des extraits de code ciblés. Il est possible de structurer et d'optimiser ce processus en suivant les étapes suivantes[27].

- 1.Pré-traitement Dans la première phase du processus de détection de clone, le domaine ciblé pour comparaison est déterminé en partitionnant le code source de manière à ne conserver que les parties intéressantes. Les parties non "intéressantes" sont écartées.

  Il est plus simple de rechercher des similarités sur des portions de code qui ont été factorisées et qui contiennent un volume de code réduit [4]. La recherche de similarités sur un large volume contenant des morceaux de codes sans liens structurels apparents est plus complexe. La mise en évidence des différents composants permet d'identifier plus facilement les éléments communs.

- 2. Transformation Les unités de comparaison sont transformées en une représentation intermédiaire adéquate. La transformation peut couvrir une large gamme allant de la suppression de commentaires, de tabulations à une transformation plus complexe telle que la compilation et la décompilation de code source [27].

- 3.Normalisation La normalisation des identifiants, la réorganisation du code source et des modifications structurelles peuvent être effectuées.
- 4.Détection des correspondances Les unités de comparaison issues de la transformation sont comparées les unes aux autres afin de détecter des paires de correspondances.
- 5 Filtrage post-traitement Dans cette phase, les faux positifs sont écartés.
- 6.Agrégation Les paires de clones sont agrégées (clusters, groupes, classes ou cliques) afin de réduire le volume de données et de faciliter l'analyse.

2.3 Algorithmes pour la détection de similarités

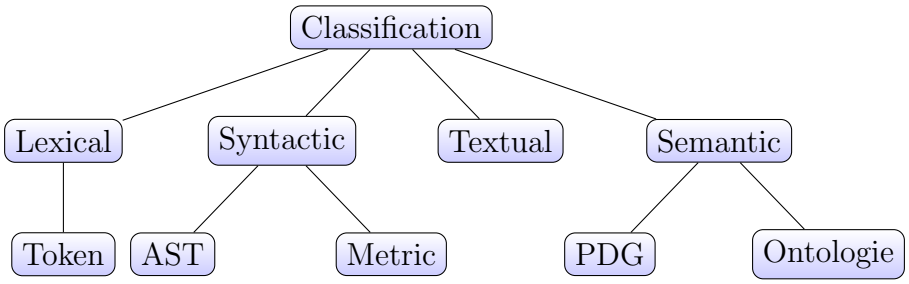

Figure 2: Classification des algorithmes de détection de similitudes

Dans cette section, sont listées plusieurs approches pour la recherche de similarités au sein de code source. Ces approches sont accompagnées de quelques points caractéristiques intéressants.

2.3.1 Les approches textuelles

- La comparaison de fragments de codes se fait par unités lexicales et par ligne. Donc, les opérations de mises en page influencent les résultats [28].
- Peu de transformations (Normalisation) sont effectuées et, dans la plupart des cas, le code source est directement utilisé lors de la comparaison [26].
- On peut utiliser des techniques de normalisation telles que le Dynamic Pattern Matching (DPM) pour supprimer les commentaires, les espaces, les tabulations et les sauts de ligne [26].

- Il est possible de générer des empreintes sur base de sous-séquences de codes.
- Des techniques telles que Latent Semantic Indexing (LSI) peuvent être utilisées pour détecter les similarités en se basant sur les identifiants et les commentaires contenus dans les lignes de code [26]. L'idée est que les mots tendent à avoir la même signification dans le même contexte.
- Les techniques d'analyse textuelle sont fiables pour détecter les clones exacts et génèrent peu de faux positifs [35].
- Elles sont aussi plus faciles à mettre en place et elles sont indépendantes des langages de programmation. Elles permettent de trouver les clones de Type-1 [31].

Outil	Transformation	Representation du code	Méthode de comparaison	Complexité	Périmètre/Granularité	Types de Clone détactable
Dup	Remove whitespace and comments	Parameterized string matches	Suffix-tree based on token match ing	O(n+m) where n is number of input lines and m is number of matches	Token of lines	Type-1 Type- 2
Duploc	Removes comments and all white space	Sequence of lines	Dynamic Pattern Matching	O(n2) where is n is number of input lines	Line	Type-1 Type- 2
NICAD	Pretty-printing	Methods (Segment sequences)	Longest Common Subse quence (LCS)	O(n2) worst case time and space	Text	Type-1 Type- 2 Type-3
SDD	No transformation	inverted index and in dex	N-neighbor dis tance	O(n)	Chunks of source code	Type-1 Type- 2 Type-3

Table 1: Exemple d'outil pour l'approche textuelle [31]

2.3.2 Les approches lexicales

- Le code source est transformé en un flux de token (tokenization) via un analyseur lexical.
- Le flux est ensuite scanné afin de trouver des sous-séquences de tokens similaires.
- On conserve une correspondance avec le code originel afin de faire le lien avec les clones.
- Cette technique permet d'identifier des clones dans des programmes incomplets ou incorrects.
- La détection de clones se base sur les arbres ou des tables de suffixes. On recherche les clones et les sous-chaînes similaires sur des chemins partant de la racine vers les feuilles de l'arbre [24].
- La disposition et l'agencement du code n'affectent pas le processus de détection.
- La technique est adaptable à d'autres langages.

- Le taux de rappel est plus important et il est possible d'avoir beaucoup de faux positifs.
- Cette technique peut être peu efficace en cas de mauvaise délimitation des frontières.

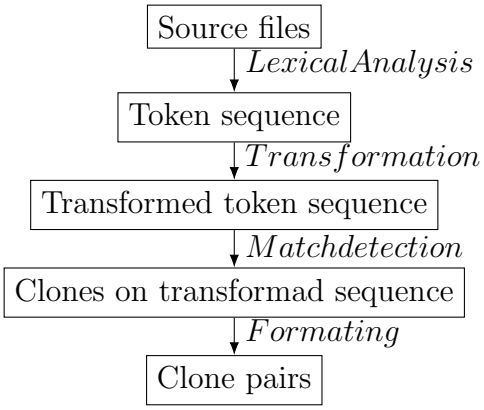

Figure 3: Token based Clone detection process [15]

2.3.3 Les approches basées sur les métriques

- On part du fait que les fragments de codes similaires engendrent des valeurs métriques similaires (complexité cyclomatique). Les clones sont des portions de code pour lesquelles les valeurs obtenues sont égales ou similaires [28].
- On travaille sur un panel de métriques pour augmenter l'intervalle de confiance. Le choix des métriques est donc primordial.

Métriques de Halstead Les mesures de complexité de Halstead fournissent une mesure quantitative de la complexité du code. Les métriques de Halstead ont, comme base de calculs, les opérandes et les opérateurs présents dans les unités de codes. Il existe plusieurs métriques de Halstead et elles se basent toutes sur le nombre total des opérateurs uniques (n1), le nombre total des opérateurs (N1), le nombre total des opérandes uniques (n2), le nombre total des opérandes (N2).

Sur base de ces chiffres, on peut calculer la longueur du programme$N = N1 = N2$, la taille du vocabulaire$n = n1 + n2$, le volume du programme$V = Nlog2(n)$, le niveau de difficulté $D = (n1/2)(N2/n2))$. Cette métrique est très sensible à l'ajout et à la suppression de morceaux de code ainsi qu'à la réécriture d'expression. De plus, on ne peut pas exclure le fait que des morceaux de code qui ne présentent aucune similarité aient pourtant des valeurs de métrique identiques.

Métrique de complexité cyclomatique

La métrique de McCabe est indépendante du langage de programmation et représente l'ensemble des instructions d'un programme. Cette métrique est calculée sur le graphe de contrôle de flux. Les nœuds représentent les instructions, les arêtes représentent le séquençage des instructions.

La complexité vaut: $EN + 2P$

E : Le nombre d'arêtes

V : Le nombre de nœuds

P : Le nombre de composantes connexes

Cette complexité est sensible à l'ajout de codes non utiles. Une complexité dynamique serait plus judicieuse, sans compter sur le fait que cette complexité reste généralement faible (20 unités) et présente donc un caractère discriminant assez faible. Il existe encore d'autres métriques telles que l'index de maintenabilité qui se calcule à partir des métriques de McCabe et des métriques de Halstead.

Vecteur de comptage de noeuds ou de sous-arbres Le vecteur de comptage de nœuds est une technique fournissant le nombre de noeuds différents présents dans chaque arbre. En généralisant cette technique, on peut fournir pour chaque sous-arbre un vecteur caractéristique afin de pouvoir les factoriser en utilisant la technique de LSH. [1] La technique de vecteur de comptage est insensible aux opérations de transposition de code, mais pas à des modifications ponctuelles dans le code telles que la réécriture d'expression, l'ajout ou la suppression de fragment de code, ayant pour résultat une modification du type des sous-arbres et des vecteurs caractéristiques.

Techniques sur base des arbres de syntaxe abstraite

- Via un algorithme de transformation, le code source est transformé en une structure d'arbres.
- Les sous-arbres sont comparés les uns aux autres et ceux qui sont suffisamment similaires sont considérés comme clones.
- Le nombre de sous-arbres à comparer peut être important. Il faut donc utiliser des critères de sélection.
- AST est parfois utilisé comme résultat intermédiaire.

[1] Cette méthode de hachage diffère des techniques conventionnelles car l'espace de collisions est maximisé plutôt que d'être minimisé. LSH permet donc de hacher avec une probabilité plus forte des entrées similaires vers de mêmes espaces d'empreinte. L'espace du résultat étant plus restreint que l'espace des entrées, elle peut être utilisée pour le partitionnement de données et la recherche du voisin le plus proche.

- Le temps d'exécution est important. Cela est dû aux nombreux sous-arbres.
- La précision est plus importante dans cette technique car la structure du programme est représentée dans la structure arborescente [13]. On peut donc mieux trouver les clones de syntaxe.
- Le taux de faux positifs est bas par rapport aux autres techniques.
- Cette technique est utile quand il faut effectuer du réusinage de code (refactoring).

Arbre de suffixes basé sur la correspondance de jeton Quand le code source est représenté sous forme d'arbre de syntaxe, la recherche naïve de sous-arbres similaires impliquerait la comparaison exhaustive de tous les sous-arbres les uns avec les autres. C'est la raison pour laquelle il est nécessaire de faire appel à des techniques qui permettront de faire des discriminations et de ne sélectionner que les sous-arbres intéressants. Des techniques de hachage et de métrique peuvent être utilisées.

Il est possible d'utiliser des arbres de suffixes contenant des unités de codes qui ont été hachés afin de générer des empreintes. Différentes stratégies de hachage peuvent être envisagées afin de réduire les faux positifs [6]. Ensuite, il faut faire appel à des algorithmes de recherche de sous-chaîne. Plusieurs stratégies d'abstraction peuvent être envisagées afin de pouvoir adopter la recherche de similarité à des niveaux de détail plus ou moins importants [5].

2.3.4 Les approches basées sur les graphes de dépendance

- Le graphe de dépendance d'un programme (PDG) est une représentation du flux de contrôle ou de données.
- Des sous-graphes similaires sont considérés comme étant des clones.
- Des informations supplémentaires sur les clones peuvent être obtenues grâce au PDG.
- Tout comme dans le cas des AST, la recherche est coûteuse et dépend du langage de programmation qui est analysé [13].

Dans le graphe de la figure 4, sont représentés deux graphes de dépendances générés automatiquement, sur base de deux codes sources différents. Il est aisé d'identifier les similarités structurelles entre ces deux codes sources.

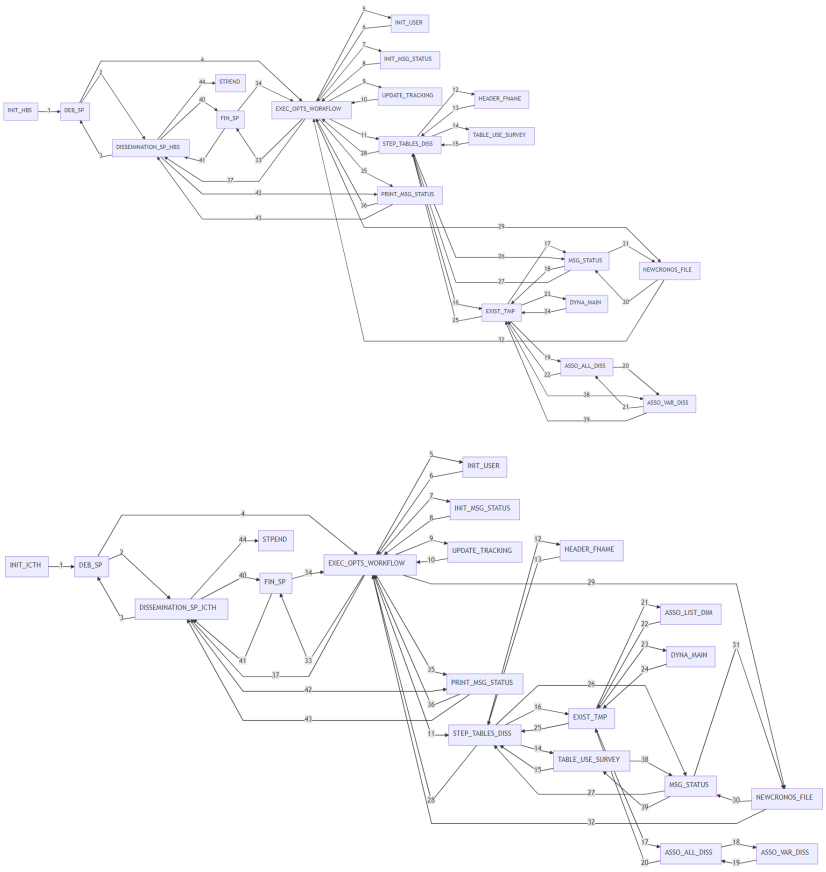

Figure 4: Génération automatique de graphes de dépendances (projet personnel)

Slicing based clone detection L'approche utilisée par cet algorithme consiste à créer le graphe de dépendance des deux portions de code à comparer. Les nœuds représentent les prédicats et les instructions. Les arêtes représentent les séquencements et les dépendances de données et de contrôle.

On partitionne le graphe en classe de structure similaire puis en partant de deux nœuds $(r1, r2)$ présentant des similarités. On recherche les graphes isomorphes en recherchant des prédécesseurs et des successeurs communs à $r1$ et $r2$.

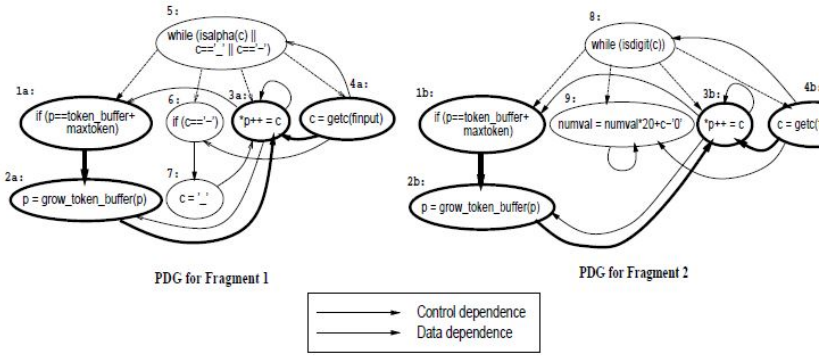

Fig. 2. Matching partial slices starting from `*p++ = c;`. The nodes and edges in the partial slices are shown in bold.

Figure 5: Correspondances partielles des découpes

K-length patch algorithm L'approche utilisée consiste à construire les graphes de dépendances dans lesquels les nœuds représentent les instructions d'affectation et les prédicats de contrôle existant dans un programme. Les arêtes représentent les dépendances entre les composants. Une arête existe entre deux composants s'il existe des dépendances en termes de séquencement ou de données. Ce qu'il faut surtout retenir, c'est que l'existence de l'arête $(v1, v2)$ se borne à l'évaluation positive de prédicat $v1$ lors de l'exécution de $v2$.

Cela signifie qu'il n'y a pas de contrainte sur le type de $v1$ et de $v2$. Cette approche permet de trouver des similarités structurelles car on recherche l'isomorphisme parmi les plus grands sous-graphes. Mais cela ne permet pas de trouver des similarités sémantiques pour des portions de code ayant été implémentés avec des approches structurelles différentes [19].

On peut citer l'outil GPLAG dont l'approche est similaire mais dans lequel les nœuds ne peuvent être que d'un seul type [23].

2.3.5 Les approches ontologiques

La manière de procéder dans cette approche vise à créer un modèle de données représentant les concepts d'un domaine et les relations entre ces concepts. La recherche d'équivalence s'effectuera sur base des informations obtenues grâce à ces concepts.

La figure [14] montre une approche dans laquelle la récupération de l'information se fait par la création de documents pour chaque méthode. Les mots-clés provenant de ces méthodes sont ensuite extraits. Ces mots-clés sont les commentaires, les chaînes de caractère et les identificateurs (classes, attributs,

méthode et paramètres). L'algorithme effectue ensuite le fractionnement de mots et extrait l'origine de chaque mot.

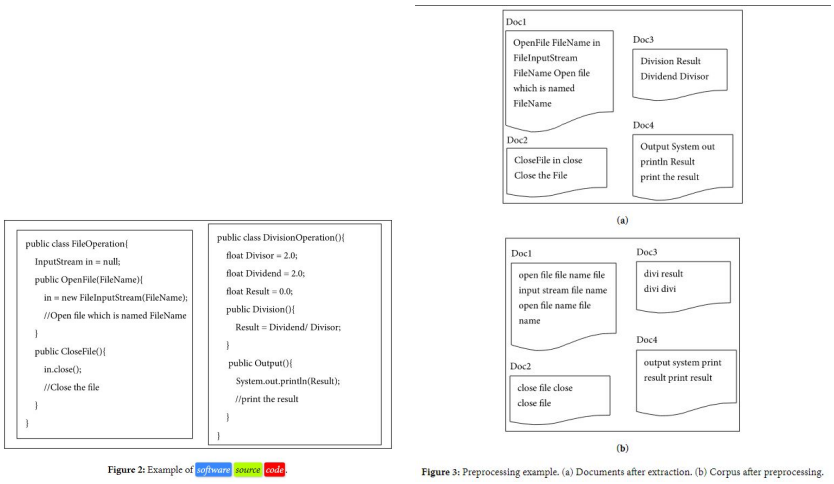

Figure 6: Exemple de code source et exemple de traitement : 1 extraction, 2 Corpus après traitement [14]

D'autres approches équivalentes consistent à trouver des similarités dans des fragments de code sur base de leur description textuelle. La source du corpus peut être Stackoverflow [34], Wikipedia [29].

Plusieurs autres approches ont été publiées dans la littérature [30], [8], et celle de Doc2vec [21] dont il est mention dans la suite de ce mémoire.

3 Algorithmes d'apprentissage automatique

Les réseaux de neurones sont des algorithmes d'apprentissage automatique (Machine Learning) générant des modèles de prédiction sur base des données qui leur sont fournies.

Dans le cas d'algorithmes d'apprentissage supervisé, le fonctionnement de l'algorithme est similaire à de l'apprentissage par essai-erreur. Pour apprendre, le système artificiel utilise, en plus des données d'entrée, les sorties correspondantes. Pour une série de N phrases d'entraînement T, on peut noter le corpus de données d'entraînement sous cette forme:

$$T = (x_i, y_i) \quad \forall i \in \{1 \dots N\} \tag{1}$$

Les données d'entrée x_i sont successivement évaluées par le système artificiel afin de fournir les sorties $f(x_i)$. L'erreur qui constitue la différence entre le résultat $f(x_i)$ obtenu et la valeur désirée y_i est prise en compte et le système est capable d'adapter son comportement.

Dans le cas d'apprentissage non supervisé, il n'est pas possible d'avoir une mesure de résultat. Les seules données disponibles sont les x_i. L'objectif des algorithmes non supervisés est, entre autres, de déterminer, grâce à l'extraction des caractéristiques présentes au sein des données, la façon dont les données sont organisées et amassées (clustered). Ces résultats obtenus dans le cas d'algorithmes non supervisés sont de types discret et qualitatif. En comparaison, les algorithmes d'apprentissage de type supervisé fournissent, eux, des résultats continus de type quantitatif.

Parmi les algorithmes d'apprentissage supervisé, on peut citer, entre autres, les régressions de type linéaire, non linéaire, les régressions logistiques, les classifications et les réseaux de neurones.

Les régressions prédisent des fonctions mathématiques censées épouser pour le mieux le comportement et la relation existant entre les données d'entrée et les données de sortie. Les régressions linéaires (linear regression) produisent des fonctions mathématiques de premier degré. Et les régressions non linéaires (logistic regression), régression linéaire multiple, tiennent compte de plus de paramètres en entrée et produisent des fonctions polynomiales.

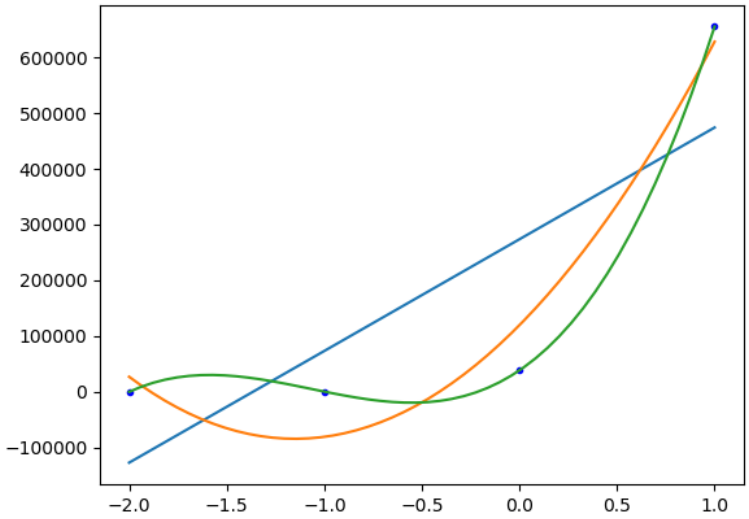

Figure 7: Exemple de régressions réalisées avec Python

3.1 Les Réseaux de neurones

Le principe de fonctionnement des réseaux de neurones artificiels est inspiré des neurones biologiques du système nerveux. Schématiquement, ces neurones biologiques sont des interrupteurs réagissant à des stimuli électriques et biologiques. Ces neurones sont hautement connectés les uns aux autres et l'information que reçoit un neurone en son entrée résulte du pré-traitement effectué par des millions de neurones en amont. Au cours des processus d'apprentissage, de nouvelles connexions interneuronales se créent permettant d'augmenter la pertinence des réponses.

Dans le cas de réseaux de neurones artificiels (Artificial neural network ANN), les neurones, aussi appelés nœuds ou perceptrons, sont interconnectés au sein d'un réseau hiérarchique à plusieurs couches.

Chaque neurone prend en entrée plusieurs signaux pondérés. On applique ensuite une fonction mathématique à la somme de ces entrées pondérées. Cette fonction dite « fonction d'activation » (Softmax function) joue le rôle d'interrupteur.

La fonction d'activation discrétise les valeurs d'entrée en deux états possibles. La distinction entre les états est fonction de la pente de la courbe. Les plages de valeurs sont généralement comprises dans les plages suivantes [0,1], [-1,1],[0,..].

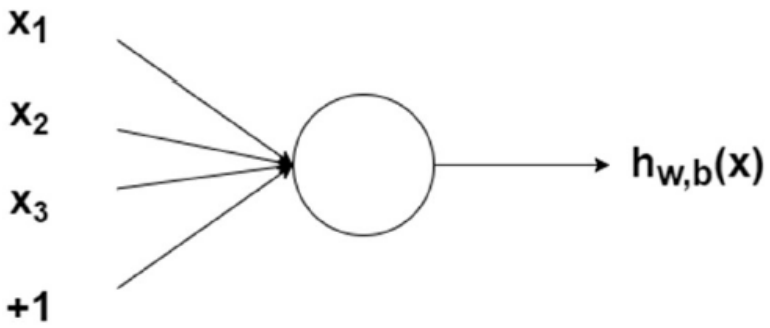

Figure 8: Exemple d'un noeud à trois entrées [32]

Mathématiquement, la somme pondérée des signaux à l'entrée d'un nœud peut être écrite de manière suivante:

$$x_1 w_1 + x_2 w_2 + x_3 w_3 + b \tag{2}$$

Dans laquelle w est le poids, il correspond à la pente de la fonction d'activation. b est le biais qui, graphiquement, correspond à un déphasage en ordonnée de la fonction d'activation.

Comme cela sera vu plus tard, le poids et le biais seront mis à jour de manière itérative lors de la phrase d'entraînement.

3.2 La structure du Réseau

Les structures des réseaux de neurones peuvent être multiples. La structure de base consiste en une couche d'entrée, une couche intermédiaire (aussi nommée « coche caché ») et une couche de sortie. Pour les cas plus complexes, plusieurs couches intermédiaires peuvent exister. Sur la figure 9, on peut voir que les nœuds d'une couche sont interconnectés aux nœuds des couches voisines.

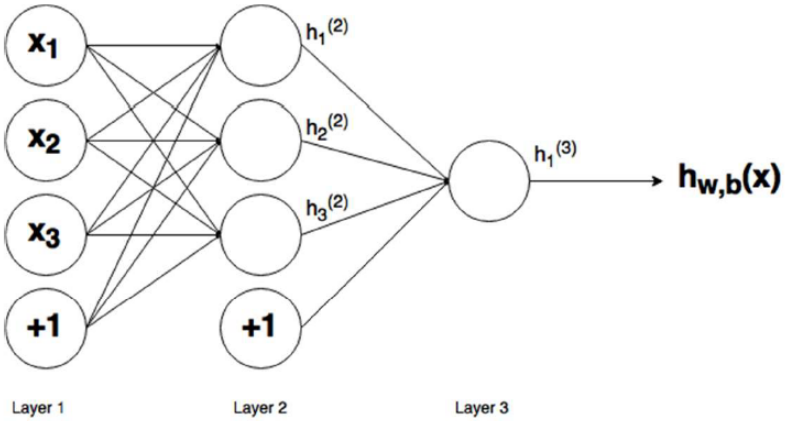

Figure 9: Exemple d'un réseau de neurone artificiel [32]

Les nœuds de la première couche sont les entrées du réseau. À partir de la seconde couche, les sorties des nœuds peuvent être calculées par la formule suivante:

$$h_1^{(2)} = f\left(w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2 + w_{13}^{(1)}x_3 + b_1^{(1)}\right) \tag{3}$$

$$h_2^{(2)} = f\left(w_{21}^{(1)}x_1 + w_{22}^{(1)}x_2 + w_{23}^{(1)}x_3 + b_2^{(1)}\right) \tag{4}$$

$$h_3^{(2)} = f\left(w_{31}^{(1)}x_1 + w_{32}^{(1)}x_2 + w_{33}^{(1)}x_3 + b_3^{(1)}\right) \tag{5}$$

$$h_{W,b}(x) = h_1^{(3)} = f\left(w_{11}^{(2)}h_1^{(2)} + w_{12}^{(2)}h_2^{(2)} + w_{13}^{(2)}h_3^{(2)} + b_1^{(2)}\right) \tag{6}$$

La notation ainsi que les équations sont celles utilisées dans le tutoriel Deep Learning de l'Université de Stanford. [7]

Dans laquelle i est le numéro du nœud à la couche $l+1$.
L'indice j est le numéro du nœud dans la couche l.
L'ordre entre i et j n'est pas forcément trivial.
Le terme x_j désigne le neurone j dans la couche l actuel.
Le terme $w_{ij}(l)$ désigne le poids affecté au neurone.
Le biais pour un nœud i à la couche $l+1$ est noté $b_i^{(l)}$.
La fonction d'activation est $f(\cdot)$.
Et la sortie du nœud i à la couche l est noté $h_i^{(l)}$.

La sortie du réseau de neurones à la couche 3 est donc définie comme:

$$h_{W,b}(x) = h_1^{(3)} = f\left(w_{11}^{(2)}h_1^{(2)} + w_{12}^{(2)}h_2^{(2)} + w_{13}^{(2)}h_3^{(2)} + b_1^{(2)}\right) \tag{7}$$

Pour utiliser ces équations en Python, elles peuvent être écrites sous forme matricielle. Le passage à une représentation matricielle est plus adapté pour le code Python qui en sera simplifié. L'utilisation des matrices permet également d'éviter l'usage de boucle, qui est à proscrire en Python lorsque la quantité de données augmente.

Pour le premier nœud de la seconde couche, la somme des entrées de la couche précédente peut être écrite de façon suivante:

$$z_1^{(2)} = w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + w_{13}^{(1)} x_3 + b_1^{(1)} = \sum_{j=1}^{n} w_{ij}^{(1)} x_i + b_i^{(1)} \tag{8}$$

Dans ces égalités, la nouvelle variable $z_i^{(l)}$ désigne le nœud i de la couche $l+1$. Et on fait la somme des n nœuds dans la couche l.

Par exemple, pour les couches 2 et 3, cette équation peut être réduite de manière suivante:

$$z^{(2)} = W^{(1)} x + b^{(1)} \tag{9}$$

$$h^{(2)} = f\left(z^{(2)}\right) \tag{10}$$

$$Z^{(3)} = W^{(2)} h^{(2)} + b^{(2)} \tag{11}$$

$$h_{W,b}(x) = h^{(3)} = f\left(z^{(3)}\right) \tag{12}$$

Dans laquelle le W est la forme matricielle du poids.

En généralisant cette équation pour toutes les couches, on obtient l'équation suivante:

$$Z^{(l+1)} = W^{(l)} h^{(l)} + b^{(l)} \tag{13}$$

$$h^{(l+1)} = f\left(Z^{(l+1)}\right) \tag{14}$$

Ce qui permet de voir que, dans le processus d'alimentation vers la dernière couche, la sortie de la couche l devient l'entrée de la couche $l+1$.

3.3 Optimisation par l'algorithme du gradient

Dans les algorithmes d'apprentissage supervisé, l'objectif est de réduire l'erreur vis-à-vis de la sortie calculée. Pour ce faire, on utilise les paires d'entrées/sorties:

$$\left\{\left(x^{(1)}, y^{(1)}\right), \left(x^{(2)}, y^{(2)}\right), \ldots, \left(x^{(m)}, y^{(m)}\right)\right\} \tag{15}$$

Dans laquelle m est le nombre d'échantillons (m phase d'entraînement) et chaque $x^{(i)}$ étant un vecteur.

Comme illustrées dans la figure suivante, ces paires d'entrées/sorties sont utilisées dans l'algorithme du gradient.

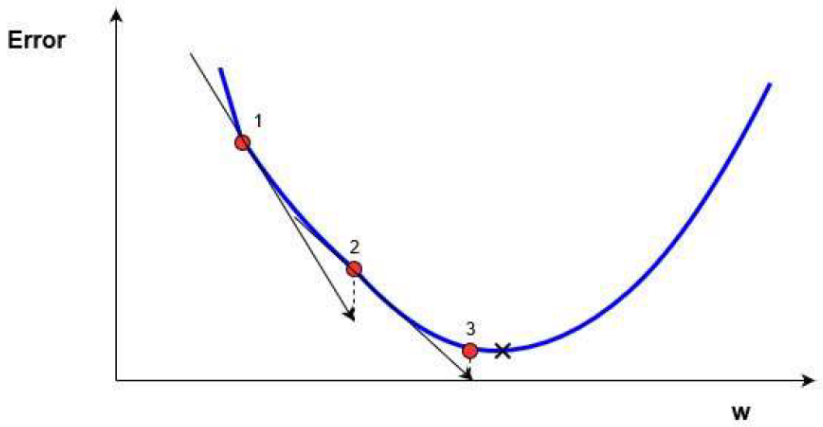

Figure 10: Graphe en une dimension pour l'algorithme du gradient [32]

La direction à suivre est donnée par la ligne droite et l'ajustement du poids peut se faire de via la formule suivante:

$$w_{new} = w_{old} - \alpha * \nabla error \tag{16}$$

Dans laquelle le w_{new} la nouvelle valeur de w (poids) w_{old} la valeur courante du w ∇error désigne la magnitude de l'erreur Le pas α correspond à l'avancement entre deux itérations. Plus il est grand, plus la convergence vers l'erreur minimum sera rapide au risque de dépasser éventuellement le point de valeur minimum. Plus on se rapproche du minimum, plus la dérivée tend vers 0. Aux abords du point d'équilibre, la valeur de w s'ajuste faiblement. Pour éviter de tendre indéfiniment vers un minimum, une valeur de seuil (précision) peut être définie.

3.4 La fonction de coût

Une des manières génériques de calculer la magnitude de l'erreur est d'utiliser la fonction de coût. Pour la paire d'entrées/sorties (x^2, y^2) à l'étape z, cette fonction de coût s'écrit:

$$J(w, b, x, y) \quad = \frac{1}{2} \left\| y^z - h^{(n_l)}\left(x^z\right) \right\|^2 \tag{17}$$

Dans laquelle z correspond à l'étape d'entraînement et $h^{(n_l)}$ correspond à la sortie du réseau de neurones de n couche (l pour « layer »). Les doubles barres verticales correspondent à la norme euclidienne.

L'ajustement des valeurs des neurones se fait de la couche de sortie vers la couche d'entrée en passant par les couches intermédiaires. C'est le principe de rétropropagation dans laquelle les neurones d'une couche l sont ajustés au travers des neurones des couches $l+1$.

Dans la figure suivante, il n'y a qu'un seul neurone dans la couche de sortie. Les neurones de la couche précédente sont ajustés via l'information delta δ provenant du dernier neurone en passant par le poids de la connexion.

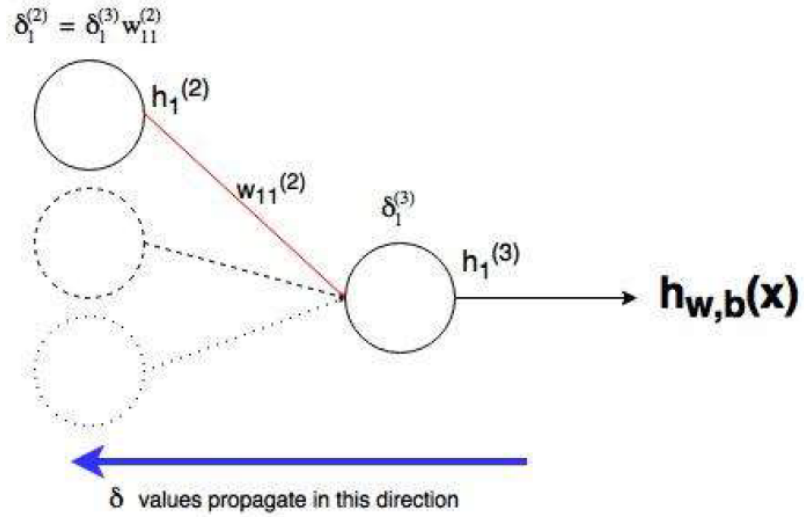

Figure 11: Illustration d'un propagation simple [32]

On peut écrire :

$$\delta_1^{(2)} = \delta_1^{(3)} w_{11}^{(2)} \varphi \tag{18}$$

Grâce aux calculs dont les détails sont en annexe, on peut trouver que:

$$\varphi = \mathrm{f}'\left(\mathrm{z}_1^{(2)}\right) \tag{19}$$

Ce qui revient à écrire de manière plus générale

$$\delta_j^{(l)} = \delta_1^{(l+1)} w_{1j}^{(l)} f'\left(z_j\right)^{(l)} \tag{20}$$

Cette information est progressivement rétropropagée de manière similaire vers les neurones des couches plus internes jusqu'à atteindre les neurones de la première couche.

Dans le graphe suivant, on peut voir que la valeur d'un neurone est ajustée sur base de trois neurones de la couche adjacente.

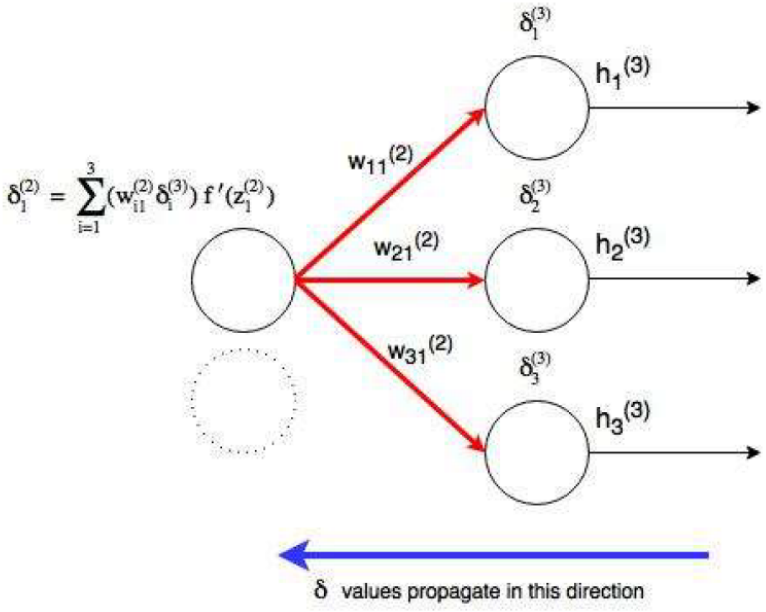

Figure 12: Exemple d'un réseau de neurone artificiel [32]

Dans cet exemple, δ peut s'écrire sous la forme:

$$\delta_1^{(2)} = \sum_{i=1}^{3} \left(w_{i1}^{(2)} \delta_1^{(3)} \right) \mathrm{f}' \left(\mathrm{z}_1^{(2)} \right) \tag{21}$$

De manière générale, pour tous les neurones du réseau, on peut écrire:

$$\delta_j^{(l)} = \left(\sum_{i=1}^{s_{(l+1)}} w_{ij}^{(l)} \delta_i^{(l+1)} \right) f' \left(z_j^{(l)} \right) \tag{22}$$

Et il est possible d'exprimer ce δ selon la fonction de coût:

$$\frac{\partial J}{\partial W^{(l)}} = h^{(l)} \delta^{(l+1)} \tag{23}$$

$$\frac{\partial J}{\partial b^{(l)}} = \delta^{(l+1)} \tag{24}$$

Une fois que la rétropropagation a été effectuée pour un échantillon, le processus est répété: on effectue le calcul des valeurs de sorties y et puis la rétropropagation s'effectue pour l'ensemble des échantillons (x^2, y^2) disponibles. Ces opérations successives vont adapter les valeurs de chaque neurone du réseau pour que l'erreur entre les sorties et les valeurs de sorties désirées soit la plus faible possible.

Suite à cette phase successive d'entraînement, le réseau est prêt à être utilisé pour de réelles prédictions : faire de nouvelles prédictions sur des entrées pour lesquelles on ne dispose pas de sorties. Dans le cadre de ce mémoire, l'étude visera à vérifier s'il est possible d'extraire et de retrouver des informations de type sémantique présentes dans des fichiers de code source. Il faudra déterminer si les codes sources sémantiquement équivalents présentent des représentations vectorielles proches.

3.5 Word2vec

Word2vec est une implémentation du plongement lexical (Word embedding [**Rong2014Word2vecPL**]) permettant de concevoir des représentations multidimensionnelles de mot [1]. Elle est utilisée dans le cadre de traitement automatique de langages.

Il existe deux implémentations de Word2vec. La première implémentation nommée CBOW (Continuous bag of words) a pour but de prédire un mot sur base de plusieurs mots de contexte. Et la seconde approche nommée Skip-Gram prédit les mots de contexte en partant d'un seul mot.

Partant du fait que des mots sémantiquement proches dans leurs contextes d'utilisation se retrouvent proches les uns des autres dans un texte. Ces mots étant proches, on peut alors fixer une fenêtre délimitant les mots de contexte à utiliser [2]. On peut noter $w_1 \dots w_m$ les m mots du contexte t sachant que $m = 2t$. Une taille de fenêtre de deux signifie qu'il y a deux mots de contexte de part et d'autre du mot cible.

Le but de la représentation neuronale des mots effectuée dans CBOW est de calculer la probabilité de prédire correctement un mot w sur base des mots de son contexte $P(w|w_1 w_2 \dots w_m)$. Cette probabilité devra être maximisée à travers le cycle d'entraînement.

L'architecture de CBOW est illustrée dans la figure suivante:

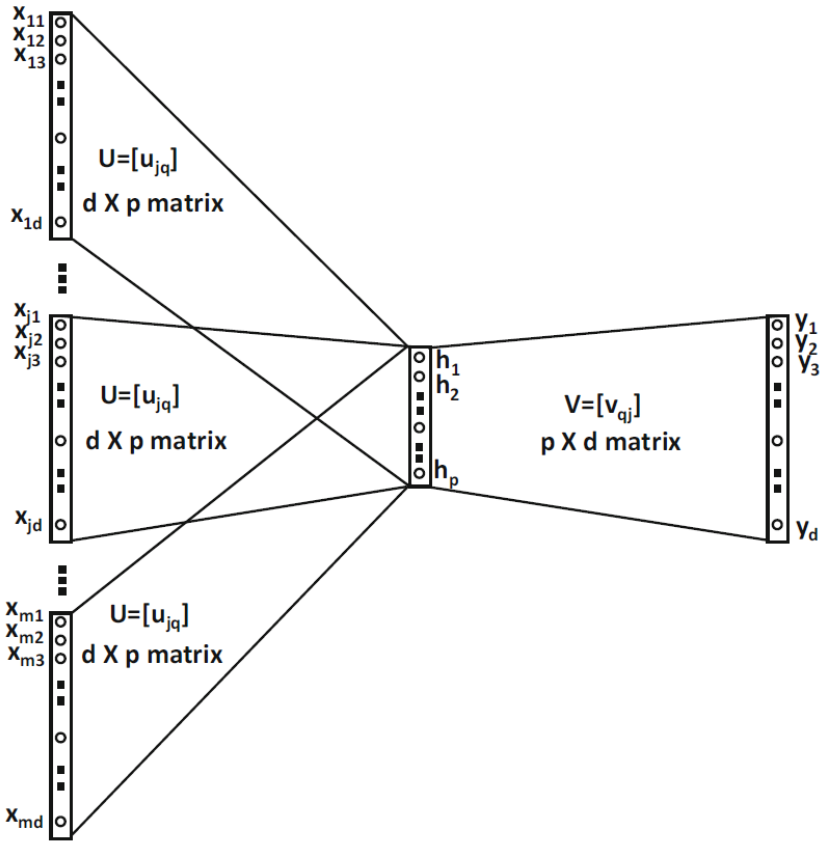

Figure 13: Le modèle CBOW de Word2vec [1]

Le réseau se compose des trois couches génériques.

Dans la couche d'entrée, chacun des m mots de contexte est représenté par un vecteur de taille d.

Les nœuds dans la couche d'entrée sont groupés en m groupe de taille d. d étant la taille de dictionnaire et m la taille de la fenêtre de mots. Chaque groupe correspond donc aux encodages $1surd$ (one-hot encoded) des mots de contexte. Chaque neurone à l'entrée peut être représenté par $x_{ij} \in \{0, 1\}$ avec $i \in \{1 \dots m\}$ étant sa position dans le contexte et $j \in \{1 \dots d\}$ étant l'identifiant du mot (sa position dans le dictionnaire).

Chacun de ces m groupes est connecté à la couche intermédiaire par des matrices U de taille dXp, où p est la taille de la couche intermédiaire. Les éléments de cette matrice u_{jq} sont les poids des neurones.

La représentation neuronale de taille p du mot j peut être notée $\overline{u}_j = (u_{j1}, u_{j2}, \dots u_{jp})$. Et pour un mot de contexte donné, sa représentation lors d'une instance spécifique sera notée $\overline{h} = (h_1 \dots h_p)$.

La sortie de la couche intermédiaire peut donc s'exprimer de manière suivante:

$$h_q = \sum_{i=1}^{m} \left[\sum_{j=1}^{d} u_{jq} x_{ij} \right] \quad \forall q \in \{1 \dots p\} \tag{25}$$

Et sous forme vectorielle, on peut l'écrire ainsi:

$$\overline{h} = \sum_{i=1}^{m} \sum_{j=1}^{d} \overline{u}_j x_{ij} \tag{26}$$

On peut noter la réduction de la dimension qui passe de d vers m et on la perte au passage de l'ordonnancement des mots de contexte. D'où le nom de la méthode.

Dans la couche de sortie par contre, on retrouve un vecteur caractéristique dont la taille est égale à celle du dictionnaire. Cela est obtenu en multipliant $(h_1 \dots h_p)$ par la fonction logistique softmax (fonction exponentielle normalisée) V de taille pXd.

Les valeurs de sortie $y_1 \dots y_d$ étant des probabilités, on obtient des nombres réels plutôt que des valeurs binaires.

Comme dans les réseaux de neurones génériques, la mise à jour des poids peut être effectuée à chaque échantillonnage à l'aide de l'algorithme de propagation inversé.

3.6 Doc2vec

Doc2vec est une extension du modèle Word2vec dans laquelle le réseau neuronal est augmenté d'un vecteur supplémentaire qui est le vecteur de paragraphe. Ce vecteur de paragraphe est pris en compte et traité de la même manière que les autres neurones. Il faut cependant noter que la taille de son dictionnaire d' peut être différente car ce texte peut provenir d'un autre cadre d'utilisation du mot cible. L'encodage de ce document passera par d' entrées binaires. Sa matrice de poids U' sera donc de taille $d'Xp$. Cela implique également qu'il n'y a pas de lien entre la mise à jour des poids de ces vecteurs de paragraphe et les vecteurs de mots. On peut dire qu'il n'y a aucune interaction entre ces deux contextes. Cependant, le calcul des valeurs de sorties ainsi que l'algorithme de propagation inversé (l'implémentation de l'algorithme du gradient) sont effectués en parallèle pour tous les vecteurs U', Uet V.

Plusieurs architectures de Doc2vec existent. Une des possibilités, en l'occurrence celle de ce mémoire, est d'utiliser uniquement des vecteurs de

paragraphe. Dans notre cas, ces vecteurs de paragraphe sont des codes sources Python.

4 Partie pratique: recherche de similarités via Doc2vec

4.1 Protocole expérimental

L'outil de recommandation de codes sources proposé dans ce mémoire repose sur les vecteurs de paragraphe (paragraph and document embeddings via distributed continious bag of words models) et se base sur l'implémentant CBOW de Doc2vec. L'objectif de cette implémentation est de déterminer s'il est possible d'identifier des codes Python sémantiquement similaires.

L'outil a été réalisé avec la version 2.7 de Python. La version 3 fut également utilisée, car plusieurs fonctionnalités provenant d'autres projets furent testées. Il a été possible d'effectuer la cohabitation et d'utiliser conjointement les deux environnements à l'aide de Miniconda.

La mise en place de l'environnement de développement a nécessité l'installation de plusieurs librairies Python. Les plus importantes pour ce projet sont listées ci-dessous. L'utilisation d'une version spécifique de Gensim [10] a été nécessaire, car la rétrocompatible n'est pas assurée pour les datasets ayant été entraînés avec des versions précédentes.

```
numpy scipy
matplotlib ipython
jupyter pandas
sympy nose
gensim==2.2.0
scikit-learn==0.21.2
beautifulsoup4==4.5.3
```

Listing 1: Liste des librairies spécifiques au projet

L'architecture du projet est représentée dans la figure 14. Elle se compose de cinq étapes principales.

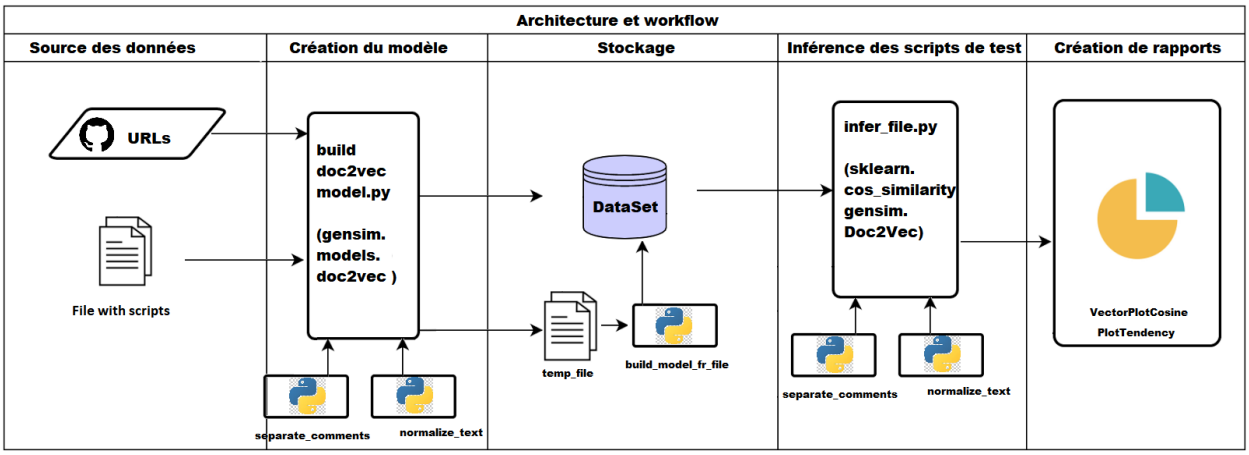

Figure 14: Déroulement des traitements

- Dans la première étape, il est possible de générer le modèle soit sur base d'un fichier contenant les URLs des scripts , soit sur base des fichiers de codes sources.

- Ensuite la création du modèle est effectuée via le script textitbuil_Doc2vec_model.py. Ce script permet de créer directement le dataset du modèle et/ou de créer des fichiers temporaires.

- Ces fichiers temporaires sont utiles afin de pallier aux failles et aux interruptions. Les dataset pouvant contenir des milliers de fichiers, il est parfois nécessaire de pouvoir interrompre le traitement. Le script textitbuild_model_fr_file.py permet de créer le dataset du modèle sur base des fichiers temporaires.

- Une fois que le dataset du modèle a été créé, il est possible de créer les vecteurs de paragraphes correspondant aux scripts de test. Cette étape s'accompagne des mêmes opérations de traitement que lors de la création du modèle.

- La dernière étape permet de visualiser les résultats à l'aide de graphiques. Des exemples de résultats sont fournis dans la suite du mémoire. L'explication détaillée des rapports se trouve également dans la suite du mémoire.

En termes de temps d'exécution, l'étape de la création du modèle est chronophage car l'ensemble des scripts doivent être traités puis concaténés. Il peut être intéressant d'avoir un aperçu des futurs résultats. Le fait d'avoir

recours aux fichiers temporaires permet de générer des modèles de taille intermédiaire (grace au script *build_model_fr_file.py*). Ces modèles peuvent alors être utilisés pour les scripts de tests. Notons que les fichiers temporaires contiennent la succession de plusieurs scripts ils sont donc beaucoup plus volumineux que le modèle final.

4.2 Les jeux de données

Afin de proposer des recommandations de codes similaires, les jeux de données suivants ont été utilisés:

Nom	**Taille (Nombre de scripts)**	**Utilité**	**Origine des scripts**
doc2vec1M.model	1,3M	Création du modèle	Projet Altair [20]
doc2vec1K.model	1K	Création du modèle	GitHub
doc2vec35K.model	35K	Création du modèle	GitHub
doc2vec50K.model	50K	Création du modèle	GitHub
doc2vec130k.model	130K	Création du modèle	GitHub
data01_2.pkl	Phase 1: 39 Phase 2: 76	Test	Mémoire GitHub

Table 2: Liste des datasets utilisés

Plusieurs datasets ont été testés pour la création du modèle. Le premier dataset est celui du projet Altair [20], il a été utilisé, car il contient 2,3 millions de scripts Python. Il n'a pas été possible de modifier ce dataset. Les URLs des scripts pyhton ne sont pas accéssible , il n'est donc pas possible de les récupérer. Ce dataset fut utilisé tel quel, car un grand nombre de scripts est nécessaire pour la construction du modèle. La vectorisation d'un vocabulaire hautement spécialisé tel qu'un langage de programmation nécessite un entraînement suffisant afin de pouvoir capturer les concepts propres au domaine [25]. De plus, l'entraînement des vecteurs de paragraphe avec Doc2vec nécessite un corpus de plus grande taille que celui utilisé pour le plongement lexical avec Word2vec. Le fait d'utiliser ce dataset permet donc de vérifier cette affirmation. Némaoins, l'ensemble des paramètres de la création de ce dernier et son contenue restent des inconnues.

Les autres datasets ont été créés sur base de scripts Python libres d'accès hébergés sur GitHub. L'obtention de ces URLs a été possible grâce à d'autres datasets du projet [20]. Aucune sélection particulière n'a été effectuée. La création du corpus Doc2vec s'effectue donc sur base des URLs de ces scripts. Une fois le code source obtenu, les commentaires et docstrings sont

supprimés, les codes sources sont ensuite normalisés en supprimant les caractères non alphanumériques ainsi que les mots vides (function words, stopwords). Ces opérations sont effectuées dans les script *separate_comments.py* et dans textitnormalize_text.py.

and	continue	except	global	lambda	pass	while
as	def	false	if	none	raise	with
assert	del	finally	import	nonlocal	return	yield
break	elif	for	in	not	true	
class	else	from	is	or	try	

Table 3: Liste des mots vides

L'entraînement sur mes jeux de données (doc2vec1K.model, doc2vec35K.model, doc2vec50K.model, doc2vec130k.model) fut effectué avec les paramètres suivants:

Paramètre	**Valeur**	**Signification**
epochs	5	Nombre d'échantillons
vector_size	300	Taille du vecteur
window	5	Taille du contexte pour un mot
min_count	10	Nombre minimum d'apparitions du mot dans l'ensemble des documents
alpha	0,05	Taux d'apprentissage
sample	1e-5	Limite au-delà de laquelle un sous-échantillonnage est effectué

Table 4: Paramètres d'entraînement

Le choix des paramètres a été guidé et ajusté en fonction des résultats obtenus. La métrique de similarité présentait dans un premier temps des valeurs anormalement faibles même pour des scripts identiques.

Le graphique (Figure 15) permet de voir l'évolution des valeurs de la métrique de similarité choisie (similarité cosinus) en fonction de la taille du dataset d'entraînement. Les datasets contenant mille, 30 mille , 50 mille, 130 mille et 2.3 millions de fichiers Python y sont représenté.

On constate une amélioration des résultats allant de pair avec la taille des datasets. Mais en ce qui concerne le dataset de taille 2.5M (courbe

pleine magenta), tous les points (même les codes non similaires) présentent des valeurs de cosinus plus élevées. De ce fait, les écarts entre les bonnes et les mauvaises prédictions sont moins grands. Finalement le dataset doc2vec130K.model a été conservé. Envieront 8 heures furent nécessaire pour la création de ce dataset, raison pour laquelle, d'autres datasets plus grand ne furent pas générés.

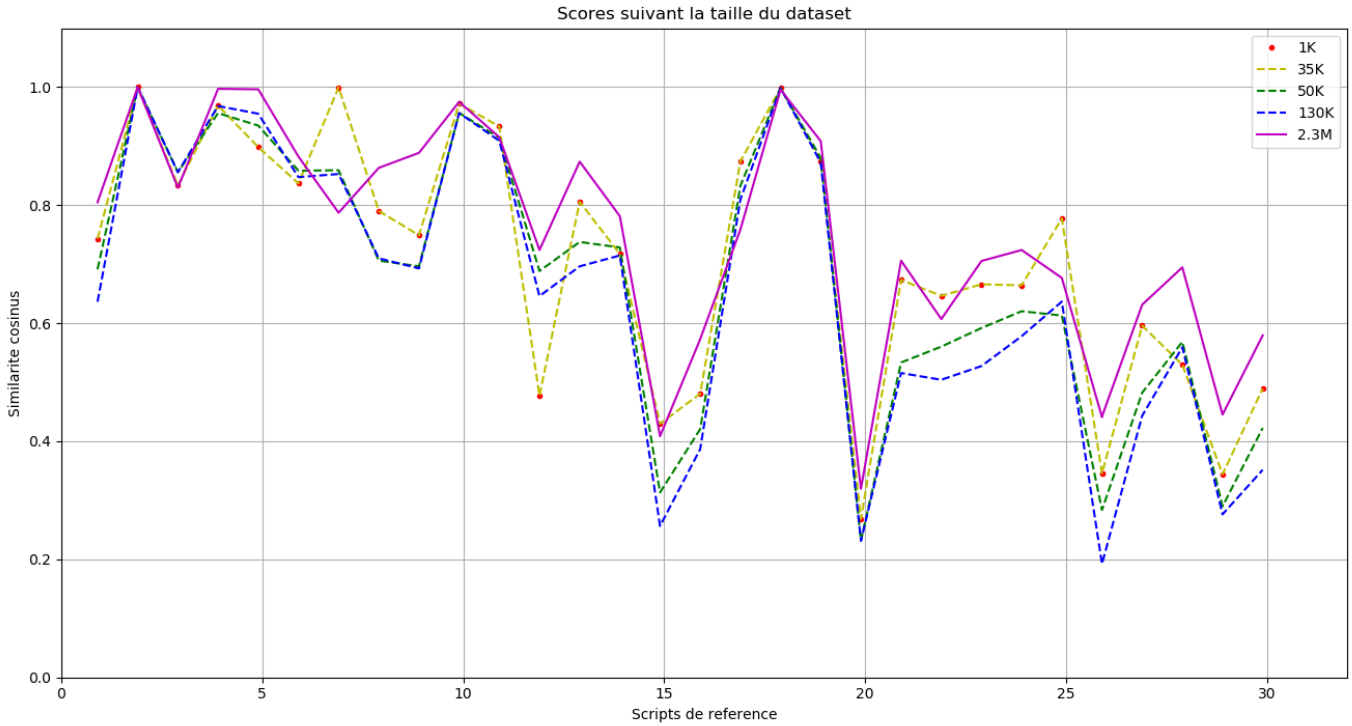

Figure 15: Similarité cosinus des clones selon la taille du dataset

Un dataset de test a également été utilisé. Il contient 39 scripts personnels (phase 1) et 35 scripts provenant de Github (Phase 2). Les 35 scripts ont été manuellement sélectionnés car elles implémentent les algorithmes suivant:

Noms des scripts	Fonctionnalilé
Karp_rabin.py	Algorithme de recherche de sous-chaîne créé
Gayle-Shapely.py	Algorithme pour le problème des mariages stables
FFT.py	Fast fourier transform
Red-Black-Trees.py	Algorithme pout l'arbre rouge et noir
Bresenham.py	Algorithme de tracé de segment
BubbleSort.py	Algorithme du tri à bulles
Depth_first-search.py	Algorithme de parcours en profondeur
Breadth_first-search.py	Algorithme de parcours en largeur
Floyd_cycle-detection.py	Algorithmique de déctection de cycle
QuickSort.py	Algorithme de tri
Dijkstra.py	Algorithme pour résoudre le problème du plus court chemin
FordBellman.py	Algorithme de calcule des plus courts chemins dans un graphe
Flood_fill-Algorithm.py	Algorithme de remplissage par diffusion
kruskal.py	Algorithme de recherche d'arbre recouvrant de poids minimum

Table 5: Algorithmes implémentés dans le 35scripts (Phase 2)

Ces 74 scripts se trouvent sur le répertoire GitHub de ce mémoire[12].

Ce Dataset de test contient plusieurs scripts sémantiquement similaires. Le nom des scripts similaires commence par les deux mêmes chiffres, mais leurs implémentations sont légèrement ou complètement différentes.

Nom du script	Modifications apportées
03_101_ContextManager.py	Variation pythonique
03_102_ContextManager.py	Saut de ligne
03_103_ContextManager.py	Renommage de la variable 'f' en 't'
03_104_ContextManager.py	Renommage de la variable 'f' en 't' et word en 'tokens'
03_105_ContextManager.py	Transposition de morceaux de code

Table 6: Exemple de scripts similaires

Nom du script	**Implémentation**	**Modifications effectuées**

Code01_1.py	condition = True
	if condition:
	x=1
	else:
	x = 0
	print(x)
		N/A (Scrips de référence)

Code01_101.py	conditions = True
	if conditions:
	xx=1
	else:
	xx = 0
	print(xx)
		Renommage de la variable 'condition' en 'conditions', 'x' en 'xx'
		et Inversion de True en False

Code01_102.py	conditions = False
	if conditions:
	xx=0
	else:
	xx = 1
	print(xx)
		Renommage de la variable 'condition' en 'conditions' et 'x' en 'xx'

Code01_2.py	condition = True x = 1 if condition else 0 print(x)	Variation pythonique

Code01_3.py	condition = False
	if condition:
	x=0
	else:
	x = 1
	print(x)
		Inversion de True en False

Code01_4.py	cond = True
	if cond:
	x=1
	else:
	x = 0
	print(x)
		Renommage de la variable 'condition' en 'cond'

Table 7: Exemple de scripts similaires

La création des vecteurs de tests se fait à la 4 ème étape de la figure 14.

L'optimum pour la création des vecteurs de tests s'est effectué avec une valeur de pas de 0.025 et en 500 étapes d'entraînement. Au-delà de cette valeur, il n'y a pas d'amélioration significative. Les quelques lignes de codes ci-dessous permettent d'illustrer cette opération.

```
r = requests.get(line)
if r.status_code == 200 :
  print ('200...')
  sys.stdout.flush()
  user_doc = r.text
  code, _ = separate_code_and_comments(user_doc,"user doc")
  normalized_code = normalize_text(code, remove_stop_words=False
    , only_letters=False, return_list=True)
  model.random.seed(0)
  model.init_sims(replace=False)

  user_vector = model.infer_vector(doc_words=normalized_code,
    steps=500, alpha=0.025)
```

Listing 2: Étapes pour la création des vecteurs de tests

Pour un script donné, la création du vecteur se fait donc en quelques étapes:

- Vérification de l'URL
- Optention du script associé à l'URL
- Premier traitement du script :separate_code_and_comments
- Second traitement du script : normalize_text
- Infère le vecteur pour le script

Cette opération est effectuée pour chaque ligne du fichier contenant les URLs fichier de test. Le Dataset est ensuite créé pour être utilisé dans la dernière étape.

4.3 Analyse des résultats (Phase 1)

Afin de générer un graphique permettant de visualiser les scripts similaires, tous les 39 scripts de cette première phase ont étés comparer les uns aux autres.

Grâce aux vecteurs générés dans l'étape précédente, on peut calculer la similarité cosinus du vecteur d'un script dit de référence par rapport à tous les autres scripts de tests. Cette valeur est proche de 1 pour les scripts similaires. Pour les scripts indépendants et complètement opposés elle vaut respectivement 0 et -1.

$$\cos\theta = \frac{A \cdot B}{\|A\| \cdot \|B\|}$$

Les résultats sont visibles dans la figure suivante.

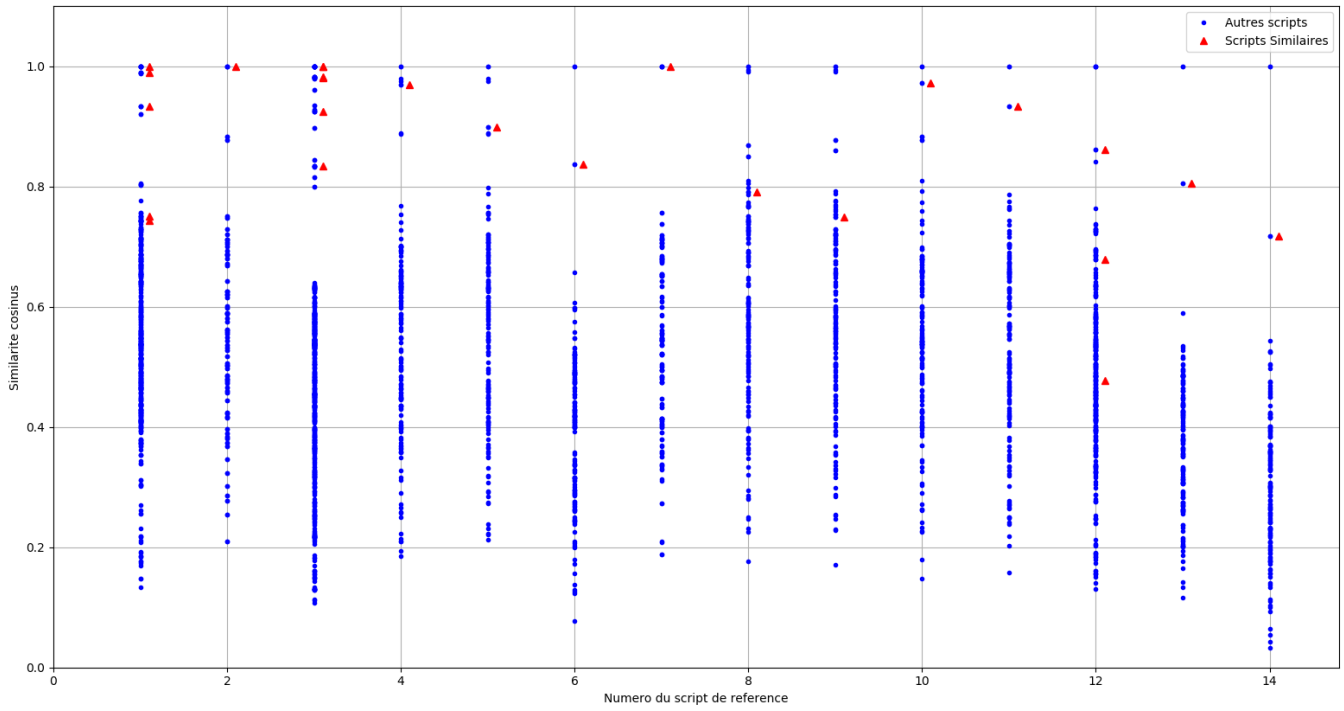

Figure 16: Résultats pour les 14 premiers groupes de scripts

Dans ce graphique, la valeur du cosinus est portée en ordonnée. L'abscisse

correspond aux scripts de référence. Les triangles rouges correspondent aux scores obtenus par les scripts sémantiquement similaires. Ils ont étés légèrement décalés vers la droite. Les points en bleu correspondent aux scores obtenus par tous les autres scripts. Les scripts de références sont les points bleus dont le cosinus vaut 1.

Pour les scripts du groupe 01 et 03, par exemple, il existe plusieurs scripts similaires. On constate que le score obtenu par ces scripts diminue lorsque davantage de modifications sont effectuées. Les tableaux (Tableau 8 et Tableau 9) listent différences présentes dans chacun des scripts. On constate que la transposition et la duplication de portion de code n'altèrent que légèrement le score. Le renommage par de mots externes au corpus d'entraînement, par exemple des mots en français, dégrade plus fortement le score.

À titre d'exemple les scripts du groupe 3 sont dans le tableau 10 si dessous:

Nom du script	**Similarité Cosinus**	**Modifications effectuées**
01_1.py	1.	N/A
01_2.py	1.	Variation pythonique
01_3.py	0.98807779	Inversion de True en False
01_4.py	0.96479907	Renommage de la variable 'condition' en 'cond'
01_102.py	0.9664449	Renommage de la variable 'condition' en 'conditions' et 'x' en 'xx'
01_101.py	0.95182812	Renommage de la variable 'condition' en 'conditions', 'x' en 'xx' et Inversion de True en False

Table 8: Modifications effectuées sur le script 01_1.py

Nom du script	**Similarité Cosinus**	**Modifications effectuées**
03_1_ContextManager.py	1.	NA
03_101_ContextManager.py	1.	Saut de ligne
03_102_ContextManager.py	0.9839413	Renommage de la variable 'f' en 't'
03_104_ContextManager.py	0.97853866	Transposition et duplication de morceaux de code
03_103_ContextManager.py	0.93021827	Renommage de la variable 'f' en 't' et 'word' en 'tokens'
03_2_ContextManager.py	0.9900979	Variation pythonique
03_105_ContextManager.py	0.79373805	Renommage de la variable 'count' en 'nombre'

Table 9: Modifications effectuées sur le script 03_1_ContextManager.py

Nom du script	Implémentation	Modifications effectuées
Code03_1_ContextManager.py	f = open('test.txt', 'r') file_contents = f.read() f.close() words = file_contents.split(' ') word_count = len(words) print(word_count)	Script de référence
Code03_2_ContextManager.py	with open('test.txt', 'r') as f: file_contents = f.read() words = file_contents.split(' ') word_count = len(words) print(word_count)	Variation pythonique
Code03_101_ContextManager.py	f = open('test.txt', 'r') file_contents = f.read() f.close() words = file_contents.split(' ') word_count = len(words) print(word_count)	Saut de ligne
Code03_102_ContextManager.py	t = open('test.txt', 'r') file_contents = t.read() t.close()l I word = file_contents.split(' ') word_count = len(word) print(word_count)	Renommage de la variable 'f' en 't'
Code03_103_ContextManager.py	t = open('test.txt', 'r') file_contents = t.read() t.close() tokens = file_contents.split(' ') token_count = len(tokens) print(token_count)	Renommage de la variable 'f' en 't' et word en 'tokens'
Code03_104_ContextManager.py	f = open('test.txt', 'r') file_contents = f.read() words = file_contents.split(' ') word_count = len(words) print(word_count) f.close() f = open('test.txt', 'r') file_contents = f.read() words = file_contents.split(' ') word_count = len(words) print(word_count) f.close()	Transposition et duplication de morceaux de code
Code03_105_ContextManager.py	f = open('test.txt', 'r') file_contents = f.read() f.close() word = file_contents.split(' ') nombre = len(word) print(nombre)	Renommage de la variable 'word_count' en 'nombre'

Table 10: Exemple des scripts similaires du groupe 3

Pour l'ensemble des scripts, les versions pythoniques [2] ne sont pas toujours détectées correctement c.-à-d. d'autres scripts obtiennent de meilleurs résultats. Dans le tableau 11, se trouvent deux exemples implémentant la même fonctionnalité et qui ne sont pas détectés comme étant des scripts similaires. Dans le tableau 12 se trouve les scripts ayant obtenue les meilleures notes de similarité avec le script Code08_1_Multby2.py. On constate que les scripts utilisant des listes ainsi que le mot clef 'for' se placent en tête des résultats.Il faut noter la présence de l'affectation 'x=1' également et en fin de liste la présence de 'condition =True' et 'if condition'. Le script Code08_2_Multby2.py arrive finalement à la dixième position.

Lot	**Script de référence**	**Script similaire**
8	x=[1, 2, 3, 4, 5, 6] for idx in range(len(x)): print(x[idx] * 2)	x=[1, 2, 3, 4, 5, 6] [print(element * 2) for element in x]

9	x=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
	for idx in range(len(x)):
	if x[idx]%2==0:
	print(x[idx] * 2)
		x=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
		[print(element * 2) for element in x if element % 2 == 0]

Table 11: Scripts similaires incorrectement détectés

[2] Pythonique: qualificatif d'un script python sémantiquement similaire, bien conçu, car respectant les règles d'écriture et plus facilement compréhensible d'autres développeurs.

Nom du Script	Score	**Code source**
Code08_1_Multby2.py	1.0	x=[1, 2, 3, 4, 5, 6] result = [] for idx in range(len(x)): print(x[idx] * 2)

Code09_1_MultByMod2.py	0.988282	x=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
		result = []
		for idx in range(len(x)):
		if x[idx]%2==0:
		print(x[idx] * 2)

Code08_2_Multby101.py	0.961423	x=[1, 2, 3, 4, 5, 6] result = [] for element in range(len(x)): print(x[element] * 2)

Code11_1_FindArrayInString.py	0.959834	arr = [”apples”, ”oranges”, ”bananas”, ”grapes”]
		s = ”cherries”
		found = False
		size = len(arr)
		for i in range(0, size):
		if arr[i] == s:
		found = True
		print(found)

Code18_2_MultipleMix.py	0.955214	x=[1, 2, 3, 4, 5, 6]
		result = []
		for idx in range(len(x)):
		print(x[idx] * 2)
		x=[1, 2, 3, 4, 5, 6]
		[print(element * 2) for element in x]
		#———
		x=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
		result = []
		for idx in range(len(x)):
		if x[idx]%2==0:
		print(x[idx] * 2)
		x=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
		[print(element * 2) for element in x if element % 2 == 0]

Code18_1_MultipleMix.py	0.952863	x=[1, 2, 3, 4, 5, 6]
		result = []
		for idx in range(len(x)):
		print(x[idx] * 2)
		x=[1, 2, 3, 4, 5, 6]
		[print(element * 2) for element in x]
		#———
		x=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
		result = []
		for idx in range(len(x)):
		if x[idx]%2==0:
		print(x[idx] * 2)
		x=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
		[print(element * 2) for element in x if element % 2 == 0]

Code05_2_ListEnumerate.py	0.949763	names = ['Peter Parker', 'Clark Kent', 'Wade Wilson', 'Bruce Wayne'] heroes = ['Spiderman', 'Superman', 'Deadpool', 'Batman'] for name, hero in zip(names, heroes): print(f'{name} is actually {hero}')

Code01_1.py	0.948598	condition = True
		if condition:
		x=1
		else:
		x = 0
		print(x)

Code01_2.py	0.948598	condition = True x = 1 if condition else 0 print(x)
Code08_2_Multby2.py	0.941502	x=[1, 2, 3, 4, 5, 6] [print(element * 2) for element in x]

Table 12: Script de multiplication par 2

Dans le graphique (Figure 16), la série des deux derniers points à droite correspond à des scripts utilisés pour l'élaboration de ce mémoire (plot_linear.py (Figure 7) , infer_file.py (Figure 14)) On constate qu'ils parviennent facilement à se différencier des autres scripts et que leurs clones sont toujours en premières positions.

Si dessous, se trouvent quelques résultats d'un test effectué sur base du dataset de test de taille 20 miles scripts python. Des résultats pris au hasard montrent plusieurs scripts, provenant de dépôts différents, dont les noms de fichiers sont proches.

```
dA_modified.py
[dA_modified.py',DenoisingAutoEncoder.py',dA.py',AutoEncoder.py
    ',denoising_autoencoder.py',denoising_autoencoder.py',
    denoising_autoencoder.py',denoising_autoencoder.py',
    autoencoder.py',RAE.py',MultiLayerPerceptron.py',
    multiplelayer.py',mlp.py',ae.py',OutputLayer.py',deepMLP.py',
    multiLayerMLP.py',mlp_maxout.py',Dropout.py',Mlp.py',MLP.py']
[1.0, 0.7999, 0.7642, 0.7474, 0.7335, 0.7334, 0.7334, 0.7258,
    0.7199, 0.6733, 0.6731, 0.6718, 0.6678, 0.6644, 0.6556,
    0.6513, 0.6444, 0.6393, 0.637, 0.6338, 0.6333]

tools.py
[tools.py',metrics.py',metrics.py',evaluate_classifier.py',
    result_util.py',rand.py',conf_matrix.py',conf_matrix.py',
    conf_matrix.py',bayes.py',evaluation_crossvalidation_results.
    py',evaluation_crossvalidation_results.py',metric.py',metric.
    py',evaluate_svm_light_run.py',classifier.py',classifier.py',
    test_evaluation.py',fscore.py',compute_statistics.py',
    evaluate_seg.py']
[1.0, 0.6231, 0.6228, 0.6184, 0.6037, 0.5894, 0.5636, 0.5636,
    0.5597, 0.5578, 0.556, 0.556, 0.5555, 0.5555, 0.5511, 0.5507,
     0.5507, 0.5472, 0.5458, 0.5443, 0.5436

caviar.py
[coffecream.py',caviar.py',classytouch.py',classytouch.py',
    rawine.py',red-grey.py',rangy.py',rawine.py',rawwine.py',
    rangy.py',rangy.py',rangy.py',rangy.py',colors.py',default-bf
    .py',ranger-red-yellow.py',ranger-red-yellow.py',finally.py',
    mydefault.py',text.py',text.py']
[1.0, 1.0, 0.9718, 0.9708, 0.9704, 0.9704, 0.9568, 0.9561,
    0.956, 0.9538, 0.9538, 0.9538, 0.9538, 0.9534, 0.9152,
    0.9061, 0.9061, 0.8983, 0.892, 0.5929, 0.5929]
```

Listing 3: Liste des librairies spécifiques au projet

4.4 Détection de scripts plus complexes

Dans le but de confirmer les résultats précédents, 37 scripts supplémentaires ont été testés dans cette seconde phase. Ces scripts implémentés par des développeurs différents contiennent plus de lignes de codes. Ce sont les scripts à partir du 15 ème groupe.

Le graphique suivant (Figure 17) permet de constater que certains clones sémantiques des nouveaux scripts ne sont pas correctement détectés.

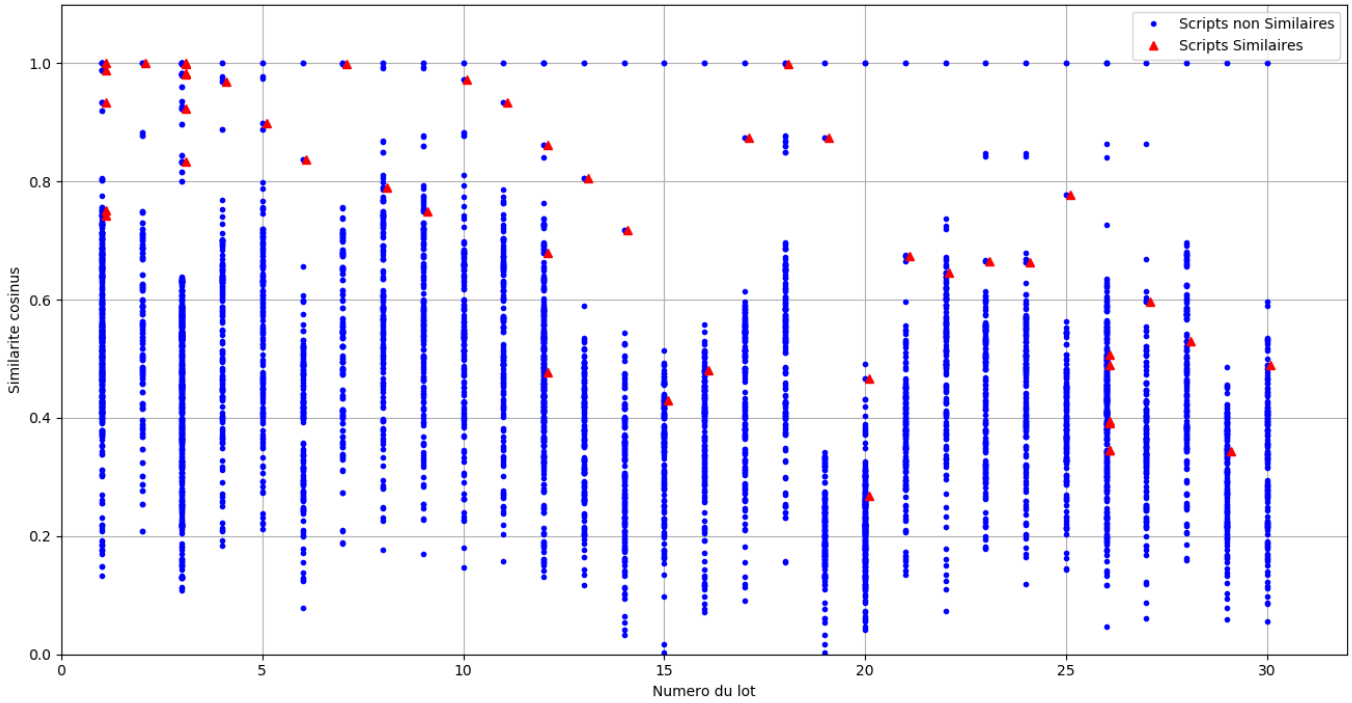

Figure 17: Similarité cosinus des clones sémantiques selon la taille du dataset

Sur les 16 groupes, 6 des clones sémantiques parviennent à être correctement détectés. En analysant le contenu de ces scripts, on constate que la moitié d'entre eux sont lexicalement proches (par exemple utilisation de liste en commune et de mot clef pyhton). Pour les scripts de groupe 17, 19 et 20, beaucoup de mots apparaissent dans le script de référence et dans la version similaire. Pour le cas du clone du script de référence '20', le score faible reste tout de même parmi les meilleurs résultats.

Scripts	Mots en commun
17_Viterbi.py	Healthy, normal, cold, def, dizzy, emission_probability, Fever, Healthy, observations, start_probability, start_probability, states, y, transition_probability, Viterbi
19_FFT.py	BIGNUM, BN_CTX, BN_GENCB, BN_GENCB, BSD, Cryptography_HAS_MGF1_MD, Cryptography_HAS_MGF1_MD, Cryptography_HAS_PSS_PADDING, Cryptography_HAS_PSS_PADDING, CUSTOMIZATIONS
20_Red_Black_Trees.py	balanced, be, black, both, by, case, check, child, children, children, class, colors, def, delete, Delete, does, each, elements, First, for, left, must, no, pushes, random, red, red-black, RedBlack, resolve, right, root, same, search, simply, step, subtree, such, swap, than, that, tree

Table 13: Mots en commun en les scripts

Le tableau 14 reprend l'ensemble des résultats. Il s'agit des scores obtenus entre tous les scripts et leurs clones sémantiques.

Nom du clone sémantique	Score	Différences avec le script de référence
01_2.py	1.	Variation pythonique
02_2.py	1.	Variation pythonique
03_101_ContextManager.py	1.	Variation pythonique
18_2_MultipleMix.py	0.99767249	Script Githug du même développeur
05_2_ListEnumerate.py	0.9923549	Variation pythonique
03_2_ContextManager.py	0.9900979	Renommage de la variable 'count' en 'nombre'
01_3.py	0.98807779	Inversion de True en False
03_102_ContextManager.py	0.9839413	Saut de ligne
04_2_ListEnumerate.py	0.98036848	Variation pythonique
03_104_ContextManager.py	0.97853866	Transposition et duplication de morceaux de code
11_2_FindArrayInString.py	0.97566933	Variation pythonique
01_102.py	0.9664449	Renommage de la variable 'condition' en 'conditions'
01_4.py	0.96479907	Renommage de la variable 'condition' en 'cond'
19_2_FFT.py	0.96338032	Script Githug d'un autre développeur
06_2_SetValue.py	0.95725658	Variation pythonique
01_101.py	0.95182812	Renommage de la variable 'condition' en 'conditions' et Inversion de True en False
08_2_Multby2.py	0.94150236	Variation pythonique
09_2_MultByMod2.py	0.93972193	Variation pythonique
03_103_ContextManager.py	0.93021827	Renommage de la varaible 'f' en 't' et 'word' en 'tokens'
07_2_Comparison.py	0.9274893	Variation pythonique
25_2_floyd_cycle_detection.py	0.92691214	Script Githug d'un autre développeur
12_102_HeapSort.py	0.88735926	Renommage de la variable 'larger' en 'largest'
28_2_FordBellman.py	0.875945	Script Githug d'un autre développeur
14_2_infer_file.py	0.86922256	Variation pythonique
24_2_breadth_first_search.py	0.85643468	Script Githug d'un autre développeur
10_2_TotalSum.py	0.84744537	Variation pythonique
13_2_plot_linear.py	0.83292274	Variation pythonique
12_101_HeapSort.py	0.82734079	Variation pythonique
17_2_Viterbi.py	0.80324704	Script Githug d'un autre développeur
26_2_QuickSort.py	0.79446782	Script Githug d'un autre développeur
03_105_ContextManager.py	0.79373805	Transposition de morceaux de code
23_2_depth_first_search.py	0.76950761	Script Githug d'un autre développeur
31_2_kruskal.py	0.72915966	Script Githug d'un autre développeur
12_2_HeapSort.py	0.72867971	Renommage de la variable 'arr' en 'array'
30_2_FFT.py	0.71117585	Script Githug d'un autre développeur
16_2_Gayle-Shapely.py	0.70435271	Script Githug d'un autre développeur
20_2_Red_Black_Trees.py	0.64543777	Script Githug d'un autre développeur
22_2_bubbleSort.py	0.64432109	Script Githug d'un autre développeur
15_2_karp_rabin.py	0.58966346	Variation pythonique
29_2_Flood_fill_Algorithm.py	0.57768123	Script Githug d'un autre développeur
21_2_Bresenham.py	0.48390023	Script Githug d'un autre développeur
27_2_dijkstra.py	0.39012898	Script Githug d'un autre développeur
20_3_Red_Black_Trees.py	0.24010497	Script Githug d'un autre développeur

Table 14: Tableau récapitulatif des résultats

5 Conclusion

En étudiant les approches existant pour la détection de similarités au sein de codes sources, il fut constaté qu'elles offraient peu de possibilité pour mettre facilement en place une solution efficace pour la détection de clones sémantiques.

Une approche se basant sur le plongement lexical implémenté via un réseau de neurones a été envisagée. La mise en place et l'implémentation de cette approche ont été facilitées par l'existence de nombreuses librairies Python dédiées à l'apprentissage automatique. Cependant, il faut parvenir à collecter un jeu de données de taille suffisante pour la construction du modèle. L'implémentation réalisée a permis de montrer qu'il est possible d'établir une relation entre des scripts Python implémentant la même fonctionnalité. Les résultats obtenus sont encourageants pour les fonctionnalités simples, mais ils ne sont pas satisfaisants pour isoler avec suffisamment de précision n'importe quels clones sémantiques.

L'obtention de résultats étant conditionnée par les jeux de données, il reste possible d'améliorer les prédictions obtenues via le contenue, le choix et le prétraitement des scripts composant ce jeu de données.

References

[1] Charu C Aggarwal. *Machine learning for text*. Springer, 2018.

[2] V Ayyadevara. *Pro Machine Learning Algorithms*. Springer, 2018.

[3] Upul Bandara and Gamini Wijayarathna. 'A machine learning based tool for source code plagiarism detection'. In: *International Journal of Machine Learning and Computing* 1.4 (2011), p. 337.

[4] Ira D Baxter et al. 'Clone detection using abstract syntax trees'. In: *Software Maintenance, 1998. Proceedings., International Conference on*. IEEE. 1998, pp. 368–377.

[5] Michel Chilowicz. 'Recherche de similarité dans du code source'. Doctoral dissertation. Université Paris-Est, 2010.

[6] Michel Chilowicz, Etienne Duris, and Gilles Roussel. 'Syntax tree fingerprinting for source code similarity detection'. In: *2009 IEEE 17th International Conference on Program Comprehension*. IEEE. 2009, pp. 243–247.

[7] Stanford University Computer Science Department. *Stanford Deep Learning Tutorial*. URL: http://deeplearning.stanford.edu/tutorial/.

[8] Yashwanth Rao Dannamaneni et al. 'Semantic code search and analysis'. Doctoral dissertation. 2014.

[9] Marguerite Francine Djomby Kome. 'Aspect mining et cohésion des classes: exploration de l'utilisation de la cohésion pour identifier les préoccupations transverses'. Doctoral dissertation. Université du Québec à Trois-Rivières, 2009.

[10] Gensim a FREE Python library. *Gensim: topic modellings for humans*. URL: https://radimrehurek.com/gensim/index.html#.

[11] Pratiksha Gautam and Hemraj Saini. 'Various code clone detection techniques and tools: a comprehensive survey'. In: *International Conference on Smart Trends for Information Technology and Computer Communications*. Springer. 2016, pp. 655–667.

[12] Github. *Projet mémoire*. URL: https://github.com/mmunsi2/python.

[13] Nils Göde and Rainer Koschke. 'Incremental clone detection'. In: *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*. IEEE. 2009, pp. 219–228.

[14] Yun He et al. 'A Structure-Driven Method for Information Retrieval-Based Software Change Impact Analysis'. In: *Scientific Programming* 2018 (2018).

[15] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 'CCFinder: a multilinguistic token-based code clone detection system for large scale source code'. In: *IEEE Transactions on Software Engineering* 28.7 (2002), pp. 654–670.

[16] Cory Kapser and Michael W Godfrey. 'Aiding comprehension of cloning through categorization'. In: *Software Evolution, 2004. Proceedings. 7th International Workshop on Principles of*. IEEE. 2004, pp. 85–94.

[17] Jitendra Yasaswi Bharadwaj Katta. 'Machine Learning for Source-code Plagiarism Detection'. Doctoral dissertation. International Institute of Information Technology Hyderabad, 2018.

[18] Rainer Koschke. 'Frontiers of software clone management'. In: *Frontiers of Software Maintenance, 2008. FoSM 2008*. IEEE. 2008, pp. 119–128.

[19] Jens Krinke. 'Identifying similar code with program dependence graphs'. In: *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*. IEEE. 2001, pp. 301–309.

[20] Lab41. *Assessing Source Code Similarity with Unsupervised Learning*. URL: https://github.com/Lab41/altair/blob/master/README.md.

[21] Jey Han Lau and Timothy Baldwin. 'An empirical evaluation of doc2vec with practical insights into document embedding generation'. In: *arXiv preprint arXiv:1607.05368* (2016).

[22] Anne-Laure Ligozat et al. *Actes de la conférence Traitement Automatique de la Langue Naturelle, TALN 2018*. 2018.

[23] Chao Liu et al. 'GPLAG: detection of software plagiarism by program dependence graph analysis'. In: *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2006, pp. 872–881.

[24] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. 'Software clone detection: A systematic review'. In: *Information and Software Technology* 55.7 (2013), pp. 1165–1199.

[25] Arpita Roy, Youngja Park, and SHimei Pan. 'Learning domain-specific word embeddings from sparse cybersecurity texts'. In: *arXiv preprint arXiv:1709.07470* (2017).

[26] Chanchal K Roy and James R Cordy. 'Scenario-based comparison of clone detection techniques'. In: *The 16th IEEE International Conference on Program Comprehension*. IEEE. 2008, pp. 153–162.

[27] Chanchal K Roy, James R Cordy, and Rainer Koschke. 'Comparison and evaluation of code clone detection techniques and tools: A qualitative approach'. In: *Science of computer programming* 74.7 (2009), pp. 470–495.

[28] Chanchal K Roy, Minhaz F Zibran, and Rainer Koschke. 'The vision of software clone management: Past, present, and future (keynote paper)'. In: *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*. IEEE. 2014, pp. 18–33.

[29] Mirco Schindler, Oliver Fox, and Andreas Rausch. 'Clustering source code elements by semantic similarity using Wikipedia'. In: *2015 IEEE/ACM 4th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*. IEEE. 2015, pp. 13–18.

[30] A-D Seriai et al. 'Mining features from the object-oriented source code of software variants by combining lexical and structural similarity'. In: *2013 IEEE 14th International Conference on Information Reuse & Integration (IRI)*. IEEE. 2013, pp. 586–593.

[31] Abdullah Sheneamer and Jugal Kalita. 'A survey of software clone detection techniques'. In: *International Journal of Computer Applications* 137.10 (2016), pp. 1–21.

[32] Dr Andy Thomas. *Neural Networks Tutorial – A Pathway to Deep Learning*. URL: https://adventuresinmachinelearning.com/category/deep-learning/neural-networks/.

[33] Michele Tufano et al. 'Deep learning similarities from different representations of source code'. In: *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE. 2018, pp. 542–553.

[34] Eran Yahav and Meital Ben Sinai. 'Code Similarity via Natural Language Descriptions'. In: ().

[35] Minhaz F Zibran and Chanchal K Roy. 'The road to software clone management: A survey'. In: *Dept. Comput. Sci., Univ. of Saskatchewan, Saskatoon, SK, Tech. Rep* 3 (2012).